Санкт–Петербургский государственный университет

КРЕСКИН Никита Владимирович

Выпускная квалификационная работа

Разработка системы мониторинга и анализа производительности серверного ПО

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная
программа СВ.5005.2016 «Прикладная математика,
фундаментальная информатика и программирование»

Научный руководитель:
кандидат технических наук, доцент,
заведующий кафедрой технологии программирования,
Блеканов Иван Станиславович

Соруководитель:
ст. преподаватель, кафедра компьютерных
технологий, Малинина Мария Анатольевна

Рецензент:
кандидат физико-математических наук, доцент,
кафедра компьютерного моделирования
и многопроцессорных систем,
Корхов Владимир Владиславович

Санкт-Петербург
2020 г.

Содержание

Введение

Информационные технологии всё глубже входят в жизнь современного человека, они используются повсеместно: в образовании, медицине, сфере развлечений, сфере услуг, и т.д.

Благодаря технологическому прорыву, плоды которого человечество особенно активно пожинает в течение нескольких последних десятков лет, сейчас уже практически невозможно представить себе человека, не пользующегося мобильным телефоном или Интернетом.

В Интернете существует множество сайтов и приложений, предлагающих широкий спектр услуг: просмотр образовательных онлайн-курсов, заказ продуктов, прослушивание аудиокниг, и прочее. Большая часть сервисов, предоставляющих эти услуги, работает "в облаке". Это означает, что для их использования нет необходимости устанавливать на устройство какие-либо приложения помимо веб-браузера, поскольку они работают на серверах - других компьютерах, которые соединены с компьютером пользователя посредством сети Интернет.

С ростом уровня потребления услуг онлайн-сервисов, растёт и нагрузка на них, и, соответственно, инфраструктура, обеспечивающая их работу: закупаются дополнительные серверы, создаются новые сервисы, существующие сервисы дробятся на более мелкие для масштабирования системы и повышения её производительности и отказоустойчивости.

Укрупнение инфраструктуры ведёт к тому, что диагностировать проблемы, возникающие во время работы приложений, становится всё сложнее. В ситуации, когда у компании есть только пять серверов, и на каждом из которых запущено по одному приложению, за этими приложениями возможно следить в ручном режиме. Но когда счёт серверов и приложений идёт на десятки, сотни или даже тысячи, вовремя заметить проблему без автоматизации наблюдения становится невозможным.

От стабильности работы веб-сервисов зависит не только досуг их пользователей: например, приложения, обеспечивающие работу банков или фондовых бирж, вовсе не могут позволить себе долгих перебоев в работе - каждая минута простоя может напрямую или косвенно стоить колоссаль-

ного количества денег. Таким образом, возможность автоматизированного наблюдения за работоспособностью системы является жизненно необходимой для бизнеса.

Постановка задачи

Целью данной работы является разработка системы мониторинга, предоставляющей функционал для слежения за работоспособностью и производительностью веб-приложений: сбором метрик, отображением собранных данных и оповещением администраторов системы в случае, если приложение работает некорректно. Для этого необходимо решить следующие подзадачи:

1. Ознакомление с существующими решениями подобных задач для выделения основных подходов, особенностей каждого решения;
2. Определение требований к разрабатываемой системе;
3. Проектирование архитектуры системы и разработка протокола взаимодействия системы с наблюдаемыми приложениями;
4. Выбор технологического стека;
5. Разработка системы;
6. Тестирование системы.

Используемые термины

1. Мониторинг — процесс проверки работоспособности приложения и сбора данных о его производительности[1];
2. Система мониторинга — комплекс программных компонентов, в совокупности решающих задачу мониторинга;
3. Метрика — множество измерений свойств или показателей приложения;
4. Алерт — периодически вычисляемая функция, применяемая к значению метрики;
5. Алертинг — процесс применения алертов и оповещения администратора системы мониторинга о достижении вычисляемой функцией заданных значений;
6. Blackbox мониторинг — подход в мониторинге, при котором метрики собираются путём выполнения запросов к внешним интерфейсам приложения;
7. Whitebox мониторинг — подход в мониторинге, при котором метрики собираются непосредственно из самого приложения (для этого, как правило, требуется интеграция в код приложения модулей, собирающих метрики);
8. Фреймворк — программная платформа, которая определяет структуру программного продукта. Набор инструментов, упрощающий разработку программного обеспечения;
9. API – набор способов взаимодействия одной компьютерной программы с другой;
10. Интеграционное тестирование – вид тестирования ПО, при котором тестируется взаимодействие различных модулей системы, их интеграция;

11. Функциональное тестирование – вид тестирования ПО, осуществляемый с целью проверки выполнения функциональных требований, то есть способности решать поставленные перед продуктом задачи;

12. Blackbox тестирование (также называемое тестированием методом «чёрного ящика») – вид тестирования ПО, при котором продукт тестируется с точки зрения внешнего пользователя, без учёта особенностей внутреннего устройства и реализации;

13. Утилита – компьютерная программа, выполняющая вспомогательные функции и решающая разного рода специализированные типовые задачи;

14. Регрессионное тестирование – тестирование, при котором проверяется работоспособность уже протестированных участков приложения.

Обзор литературы

Славек Лигус, «Effective Monitoring and Alerting: For Web Operations»[2]. Данная книга в первую очередь использовалась как теоретическое пособие, поскольку в ней раскрыты теоретические аспекты мониторинга (в частности, вопрос мониторинга крупных распределённых систем).

Майк Джулиан, «Practical Monitoring: Effective Strategies for the Real World»[3]. В этой книге приводится множество практических примеров. Ознакомление с практической стороной вопроса оказалось крайне важным для полного понимания сущности решаемой задачи.

В.П. Шкодырев, К.И. Ягафаров, В.А. Баштовенко, Е.Э. Ильина, «Обзор методов обнаружения аномалий в потоках данных»[4]. Данная статья ставит задачу поиска аномалий во временных рядах, формализует определение аномалий и описывает основные подходы к решению задачи.

А.А. Пивень, Ю.И. Скорин, «Тестирование программного обеспечения»[5]. В статье классифицированы и описаны основные методы тестирования программного обеспечения. Информация была использована при проведении нагрузочного тестирования разработанной системы.

Р.А. Нагаев, И.С. Полевщиков, «Автоматизация процесса тестирования программного обеспечения с применением Junit»[6]. В данной статье описан процесс автоматизации тестирования при помощи библиотеки Junit. Статья использовалась для написания автоматических тестов на ключевой компонент системы.

Глава 1. Сбор требований

При эксплуатации веб-приложений естественной потребностью является наблюдение за работоспособностью и производительностью системы. Так как следить за показателями приложения вручную представляется невозможным, этот процесс необходимо автоматизировать. Для этого существует мониторинг.

Главной целью мониторинга является возможность слежения за корректностью работы приложения.

Основной функционал систем мониторинга (может отличаться для разных систем):

1. Сбор метрик приложения;
2. Хранение метрик;
3. Предоставление собранных метрик по запросу;
4. Агрегация метрик и вычисление от них разного рода функций;
5. Алертинг.

Системы мониторинга следят за производительностью вверенных им приложений и серверов, позволяют администратору системы получить данные о работе того или иного интересующего его звена системы (например, узнать, насколько быстро тот или иной сервис обрабатывает запросы, или как сильно сейчас нагружен процессор на определённом сервере), а также автоматически оповещают администратора о возникших неполадках.

1.1 Анализ существующих решений

Существующие решения имеют преимущества и недостатки. Рассмотрим их подробнее и приведем сравнительный анализ.

1.1.1 Prometheus

Prometheus это самодостаточная система мониторинга, не требующая сторонних инструментов для решения основной задачи мониторинга.

Сбор данных. Для сбора данных реализована pull-модель (для сбора метрик программ, имеющих короткий жизненный цикл, поддерживается возможность буфферизации метрик перед сбором их главным сервером) — главный сервер через заданный интервал времени опрашивает все указанные в его конфигурации цели сбора данных, собирает метрики и сохраняет их.

Область применения. Prometheus собирает метрики, может их агрегировать. Система предоставляет инструменты для визуализации (построения графиков) собранных данных, а также поддерживает алертинг.
Prometheus подходит для ситуации, в которой требуется собирать много общих, не ориентированных на конкретные события метрик, а также есть потребность в whitebox мониторинге.

Модель данных. Метрики в Prometheus хранятся в виде пар ключ-значение. Пространство ключей является многомерным: каждый ключ состоит из названия метрики и множества пар ключ-значение, называемых метками. Рассмотрим пример. Представим, что у нас есть метрика
requests, которая обозначает количество запросов, совершённых за всё время к приложению «app1», установленному на сервере example-server.com. Тогда указанное значение будет кодироваться как «requests server=example-server.com, application=app1 ».

Архитектура. Prometheus состоит из следующих частей:

1. Главный модуль – сервер, который собирает и хранит метрики;
2. Клиентские библиотеки для интеграции с приложениями;
3. Push Gateway – приложение, созданное для поддержания push-модели сбора метрик (используется для мониторинга программ, имеющих короткий жизненный цикл, с которых невозможно собрать метрики в pull-модели, т.к. время их жизни может оказаться короче, чем интервал сбора метрик);

4. Менеджер алертов – модуль, отвечающий за применение к метрикам алертов и оповещение администраторов системы.

Хранение данных. На каждую метрику создаётся отдельный файл, который хранится на диске.

Вывод. Prometheus предоставляет гибкий язык запросов, позволяющий извлекать и агрегировать данные для их анализа, множество инструментов, покрывающих достаточно широкий спектр задач. Система не предоставляет возможности обрабатывать детализированные метрики, а также создаёт дополнительную нагрузку на наблюдаемые приложения из-за использования pull-модели (в связи с тем, что главный сервер Prometheus обращается к приложениям по HTTP). Ко всему прочему, масштабирование системы затруднено.

1.1.2 Graphite

Graphite представляет собой базу данных временных рядов, предоставляющую язык запросов и возможности для отображения данных. Все прочие функции реализуются отдельными внешними компонентам.

Область применения. Graphite является хорошим выбором в ситуации, когда необходимо собирать недетализиранные метрики и хранить их в течение длительного времени.

Сбор данных. Сбор данных осуществляется с использованием push-модели средствами сторонних компонентов (например, collectd[7]).

Модель данных. Метрики хранятся в виде пар ключ-значение. При этом ключи представляют собой множество пространств, идентификаторы которых разделены точками, что создаёт вложенность структуры ключей. Пример: ключ метрики из примера ключа в Prometheus будет выглядеть в Graphite следующим образом:

«stats.api-server.example-server.app1.server_requests». Такой формат упрощает поиск нужной метрики, но создаёт жёсткую иерархическую структуру в пространстве имён метрик, в связи с чем снижается гибкость языка запросов. Также добавление нового свойства метрики может повлечь за собой пересмотр множества уже существующих ключей. Например, если в ключах метрик, описывающих количество запросов, обработанных некоторым сервисом на определённом сервере, мы также захотим указывать не только конкретный сервер, но и его окружение (то есть группу серверов), нам придётся вносить изменения в ключи уже существующих метрик.

Архитектура. Сам Graphite состоит из базы данных и модуля, отвечающего за построение графиков и их отображение. Таким образом, в системе нет встроенного модуля для сбора данных или алертинга — эти функции выполняются путём подключения отдельных модулей, не связанных с Graphite напрямую.

Хранение данных. Метрики хранятся на диске. Для каждой метрики создаётся отдельный файл. Хранение происходит в формате Whisper[8].

Вывод. Graphite решает схожую с Prometheus задачу, но предоставляет менее гибкий язык запросов. Кроме того, он сложнее в установке: для того, чтобы обеспечить основной функционал системы мониторинга, необходимо использовать сторонние модули.

1.1.3 InfluxDB.

InfluxDB[9] это база данных временных рядов.

Область применения. Область применения аналогична области применения Graphite.

Сбор данных. Сбор данных осуществляется с применением push-модели средствами сторонних компонентов.

Модель данных Модель данных представляет собой набор пар ключ-значение. Идентификатор состоит из трёх составляющих: название метрики, набор меток и набор полей. И метки, и поля также являются парами ключ-значение, но при этом метки индексируются, а поля — нет. Таким образом, метки используются для поиска по данным, а поля — для хранения дополнительной информации, указанной при записи метрик в базу данных.

Рассмотрим пример. Представим структуру идентификатора метрики в виде JSON-объекта:

```
{
    "name": "weather",
    "tags": {
        "city": "Moscow"
        "metric": "temperature"
    },
    "fields": {
        "latitude": 56,
        "longtitude": 38
    }
}
```

В данном примере описан ключ метрики, показывающей погоду (в частности, такой показатель как температура) в опеделённом городе (в городе Москва). Название метрики это «weather», метки (здесь они называются «tags») это название города и тип характеристики погоды, а поля («fields») – широта и долгота. Таким образом, подобная структура может позволить нам совершить запрос, по которому мы, используя название города и интересующую нас характеристику, найдём температуру в Москве, и по полям сможем сразу узнать координаты, где был совершён замер температуры.

Важно подчеркнуть, что при хранении метки индексируются. Они могут содержать только строковые значения. А поля не индексируются и

могут содержать любые значения. Таким образом, модель данных спроектирована с расчётом на то, что поиск и агрегация будут осуществляться с использованием имени метрики и меток, а поля будут использованы для вычисления некоторых функций.

Архитектура. Основу составляет модуль хранения данных — сама база InfluxDB. Сбор данных и алертинг осуществляются сторонними средствами.

Хранение данных. Данные в InfluxDB хранятся в модифицированной версии LSM-дерева[10] — так называемом временно-структурированном дереве со слиянием (TSM-дерево[11]). База данных имеет write ahead log. После записи данные хранятся в read-only файлах.

Вывод. InfluxDB подходит для ситуации, в которой помимо метрик необходимо сохранять логи событий. Помимо этого, рассматриваемая СУБД поддерживает кластеризацию — это снимает некоторые ограничения на построение системы мониторинга при наблюдении за крупной инфраструктурой приложений и серверов.

1.1.4 OpenTSDB

OpenTSDB[12] также является базой данных временных рядов. В рассматриваемой модели присутствует модуль для визуализации данных, но отсутствует встроенное решение для алертинга.

Область применения. Область применения аналогична области применения Graphite.

Сбор данных. Сбор данных осуществляется с помощью push-модели средствами клиентских модулей (например, TCollector[13]), входящих в экосистему OpenTSDB.

Модель данных. Модель данных практически полностью аналогична модели данных Prometheus.

Архитектура. Основой OpenTSDB является один или несколько TSD (time series daemon[14]), которые подключены к общей базе данных HBase.

Хранение данных. Хранилище OpenTSDB работает на основе Hadoop[15] и HBase[16].

Вывод. OpenTSDB достаточно сложна в эксплуатации, но имеет преимущества, например, горизонтальную масштабируемость. Имеется встроенный модуль для визуализации данных, встроенный модуль для алертинга отсутствует.

1.1.5 Nagios.

Nagios[17] представляет собой систему мониторинга. Эта система была создана в 1999 году, в связи с чем для неё написано множество плагинов и расширений.

Область применения. Основная направленность работы Nagios — это обработка алертов. Nagios проводит на наблюдаемом приложении набор специальных проверок, и отправляет оповещения администратору системы о тех проверках, результат которых не соответствует ожидаемому.

Сбор данных. Реализована push-модель сбора данных.

Модель данных. Каждой проверке соответствует некоторое имя. Язык запросов или метки отсутствуют.

Архитектура. Основной единицей системы является центральное ядро — приложение, обрабатывающее проверки и записывающее их результаты в файл. Присутствует модуль для визуализации.

Хранение данных. Как таковое, хранение данных в Nagios отсутствует, за исключением хранения текущего состояния проверок. Те данные, что Nagios сохраняет на диске, складываются в один файл.

Вывод. Nagios подходит для мониторинга небольших систем, в которых будет достаточно blackbox-мониторинга и знания текущего состояния наблюдаемых приложений.

1.1.6 Sensu

Sensu это решение для мониторинга, работающее аналогично Nagios и позволяющее переиспользовать ранее проведенные проверки.

Область применения. Область применения аналогична области применения Nagios.

Сбор данных. Способ сбора данных аналогичен способу сбора данных в Nagios.

Модель данных. Модель данных также аналогична модели данных Nagios.

Архитектура. В Sensu существует основной модуль для обработки полученных метрик и обработки языка запросов. Взаимодействие с клиентскими модулями происходит через RabbitMQ[18]. Данные хранятся в Redis[19].

Хранение данных. Sensu хранит результаты проверок, историю их выполнения и текущее состояние.

Вывод. Sensu является подходящим решением для ситуации, когда текущая система уже использует Nagios, а также в целом для системы, в которой достаточным является blackbox-мониторинг.

1.2 Итоги анализа

Все приведенные системы не лишены недостатков. Возникает идея создания нового решения, учитывающего потребности и специфику современных приложений, вбираюшего в себя преимущества уже существующих систем, способного повысить продуктивность отделов эксплуатации множества IT-компаний, а также ускоряющего диагностику ошибок и позволяющего выявлять аномалии в собранных данных. Для конечного потребителя услуг сервисов это означает более высокую их доступность и скорость работы.

На основе проведённого анализа можно привести список основных требований к новой системе:

1. Система будет собирать недетализированные метрики (логи событий рассматриваться не будут);

2. Система должна собирать метрики с использованием push-модели. Выбор подхода сбора метрик является частой темой для дискуссий. В данном случае основные доводы в пользу push-модели:

   (a) снижение нагрузки на сервера с наблюдаемыми приложениями;

   (b) системе мониторинга нет необходимости знать о том, как устроен мир вокруг неё: для того, чтобы подключить к мониторингу новое приложение, не нужно менять конфигурацию системы мониторинга, достаточно просто указать приложению, куда отправлять метрики;

   (c) упрощается задача масштабирования системы мониторинга: в случае с pull-моделью, при запуске нескольких экземпляров сервера мониторинга возникла бы потребность согласовывать так называемые списки опроса (списки целей мониторинга — по сути, набор адресов приложений в сети), чтобы несколько серверов мониторинга не собрали данные с одного приложения; в push-модели этот недостаток отсутствует.

3. Для отправки метрик должен использоваться протокол UDP[20]. Основным аргументом в пользу использования UDP является его удобство и лучшая производительность в задачах, когда требуется отправлять данные, не получая ответа. Для передачи данных при помощи UDP клиент не ожидает ответа от сервера, таким образом снижая нагрузку на устройство, на котором запущено приложение. После отправки данных приложение сразу же может переключиться на выполнение своих основных задач, без необходимости держать соединение открытым до того, как сервер обработает запрос и вернёт ответ. Основным минусом использования данного протокола является его ненадёжность: нет никаких гарантий успешной доставки данных. Этим предлагается пренебречь, потому что мониторинг не должен создавать дополнительную нагрузку на наблюдаемые приложения;

4. Система должна поддерживать возможность осуществления whitebox мониторинга. Для этого потребуется поддержка интеграции клиентского модуля системы с наблюдаемыми приложениями;

5. В системе должен присутствовать встроенный модуль для алертинга. Так как алертинг представляется необходимой частью системы мониторинга, без которого система не выполняет свою основную задачу. Модуль для алертинга будет присутствовать в стандартной поставке системы;

6. Система должна масштабироваться. В современном мире IT-инфраструктура многих компаний может включать в себя тысячи или десятки тысяч одновременно работающих серверов и приложений, поэтому естественной потребностью будет создать достаточно производительную систему, чтобы она могла собирать и анализировать метрики с большого количества источников. Эту цель можно достичь только создав возможность горизонтального масштабирования;

7. Система должна быть отказоустойчивой;

8. Должен быть реализован язык запросов, позволяющий при помощи обращений к API системы мониторинга получить данные о собранных метриках, а также предоставляющий возможности для их агрегации и вычисления математических функций.

1.3 Выводы

В данной главе был проведён анализ существующих решений в области мониторинга, в соответствии с которым были сформулированы основные требования к разрабатываемой системе.

Глава 2. Разработка

В этой главе приведены архитектура системы и технологический стек, который используется для её разработки. Кроме того, описаны ключевые моменты внутренней реализации программных компонентов и внешние интерфейсы, предназначенные для организации взаимодействия с системой.

2.1 Архитектура

На рисунке 1 приведена схема с указанием основных связей между компонентами разрабатываемой системы.

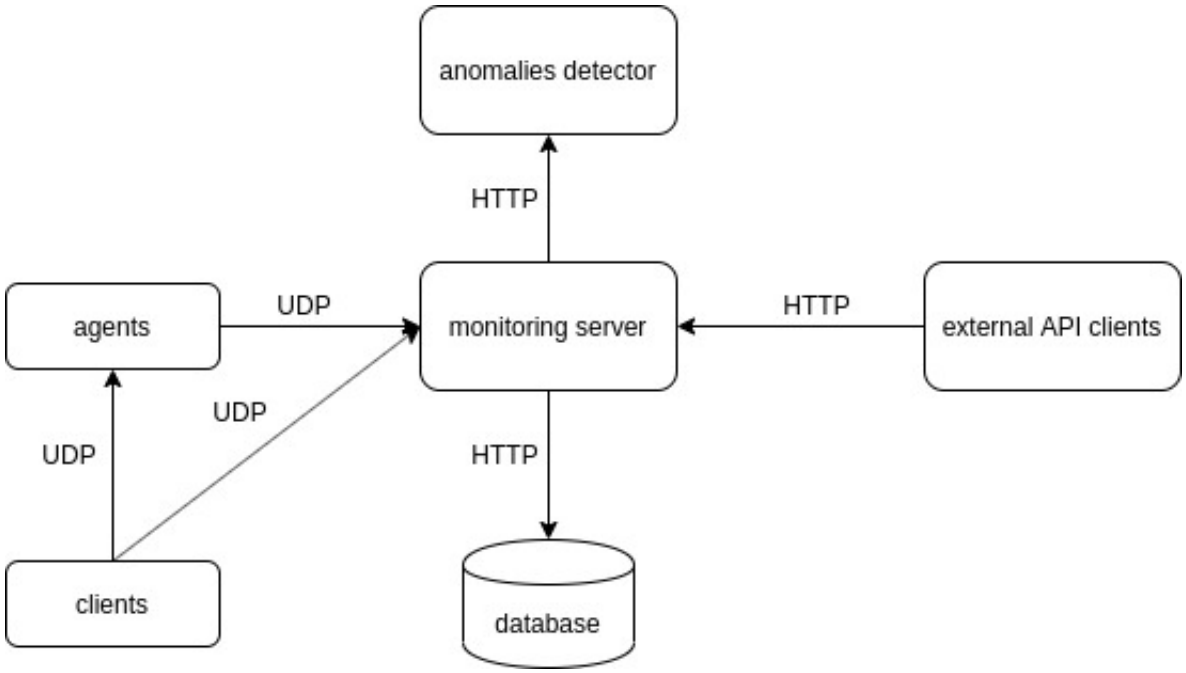

Рис. 1: Архитектура системы.

Компоненты схемы:

1. Monitoring server – ключевой элемент системы. Приложение, которое получает значения метрик с использованием протокола UDP, хранит их, агрегирует, выдаёт по запросу к API компонента;

2. Database — база данных (или кластер баз данных), в которых monitoring server хранит собранные метрики и вспомогательные данные;

3. Anomalies detector — компонент, выполняющий поиск аномалий во временных рядах;

4. Clients — приложения, отправляющие свои метрики в систему мониторинга. Они могут направлять их как напрямую, так и в агенты;

5. Agents — агенты. Приложения, которые выполняют роль буфера между клиентскими приложениями и сервером мониторинга. Не являются обязательным элементом системы. Есть три основных причины их использования:

   (a) избавление клиентских приложений от необходимости знать адрес сервера мониторинга;

   (b) возможность получения дополнительной информации. Так как агент получает все данные перед отправкой, он может добавить к данным дополнительную метаинформацию — например, адрес сервера, на котором установлено отправляющее данные приложение;

   (c) агенты снижают нагрузку на сеть и на сервер мониторинга, поскольку вместо отправки множества запросов они буфферизуют их и отправляют за один раз некоторый набор значений.

6. External API clients — приложения, обращающиеся к серверу мониторинга за собранными метриками (это могут быть приложения, выполняющие анализ статистики, или приложения для визуализации данных).

2.2 Модель данных

Модель данных в разрабатываемой системе аналогична модели данных Prometheus: каждая метрика представляет собой пару ключ-значение, где ключ имеет составную структуру вида
«metric_name label1=value1,label2=value2,. . . ,labelN=valueN », а значение является числом.

Расширять область определения значений при условии, что система должна хранить недетализированные метрики, не имеет смысла: числовые значения (с плавающей точкой) покрывают все потребности подобной системы.

Исходя из анализа проведённых решений можно выделить три типа подходящих форматов ключа:

1. Формат Graphite (имя метрики);
2. Формат InfluxDB (имя метрики, метки и поля);
3. Формат Prometheus (имя метрики и метки).

Одного имени метрики, как в Graphite, недостаточно: несмотря на то, что в имени можно закодировать дополнительные «измерения», данный формат не является гибким, поскольку добавить новое «измерение» в уже собранные данные будет затруднительно. Также такой формат создаёт жёсткие ограничения на последовательность введения данных при их поиске. В связи с перечисленными недостатками первый вариант не может быть использован.

Второй вариант даёт всю требуемую гибкость, но введение в модель полей является избыточным в рамках требований к разрабатываемой системе.

Таким образом, предпочтительнее использовать третий вариант.

2.3 Технологический стек

В данном параграфе для каждого компонента разрабатываемой системы определён набор технологий, который использован при его написании.

2.3.1 Monitoring server

Для написания основного компонента использован язык программирования Kotlin[21]. Kotlin — это язык программирования, работающий по-

верх JVM (Java Virtual Machine)[22]. Основными аргументами для выбора данного языка программирования являются:

1. Kotlin имеет все преимущества языка Java, такие как объектно-ориентированный подход, кроссплатформенность, многопоточность, безопасность (в частности, благодаря отсутствию прямого доступа к памяти посредством указателей);

2. Kotlin совместим с Java, и, как следствие, может использовать любые библиотеки, написанные на Java;

3. Kotlin имеет простой синтаксис и ограждает от множества стандартных ошибок по умолчанию. В сравнении с Java, Kotlin, например, реализует принцип null-безопасности, расширяет возможности для асинхронного программирования при помощи корутин[24] а также в общем уменьшает издержки на разработку ввиду простоты синтаксиса и готовых решений для стандартных проблем, присутствующих в Java.

В качестве фреймворка будет использован Spring Framework[25]. Данный фреймворк предлагает множество инструментов для разработки программного продукта, в частности технологии, реализующие концепцию Dependency Injection[26], инструменты для сетевого взаимодействия и инструменты для тестирования. Использование данного фреймворка значительно ускоряет процесс разработки, так как многие необходимые компоненту функции уже реализованы.

2.3.2 Database

В качестве базы данных будет использована Cassandra[27]. Это распределённая NoSQL[28] база данных, разработанная для создания высокомасштабируемых и надёжных хранилищ больших массивов данных.

NoSQL базы данных – это базы данных, имеющие существенные отличия от реляционных (SQL) баз данных, в частности:

1. Схема. Схемы SQL баз данных основываются на таблицах, в NoSQL базах данных схемы могут основываться на парах ключ-значение, документах, графах;

2. Язык запросов. Все SQL базы данных поддерживают SQL-стандарты, таким образом, они предоставляют интерфейс для получения данных при помощи SQL-запросов. В каждой NoSQL базе данных реализуется свой подход к построению запросов на получение данных, общий стандарт отсутствует;

3. Скорость. SQL базы данных реализуют принцип ACID[29]. Поддержка этого принципа создаёт дополнительные издержки. NoSQL решения чаще всего не реализуют этот принцип в полной мере, таким образом смягчая жёсткие ограничения. Также преимущество в скорости работы создаётся благодаря большей приспособленности NoSQL баз данных к горизонтальному масштабированию.

Выбор данной СУБД обоснован следующим:

1. Cassandra использует для хранения такую структуру данных как LSM-дерево. Эта структура идеально подходит для нужд системы мониторинга, поскольку операция записи в худшем случае происходит за константное время (O(1)). Это принципиально важно для хранилища метрик, ведь в систему может одновременно поступать огромное количество данных. В таком случае очень важно иметь возможность быстро их обрабатывать, чтобы не допустить потери данных (и, как следствие, искажения общей картины поведения наблюдаемых приложений);

2. Cassandra разработана для того, чтобы эффективно хранить большие массивы данных — эта потребность учтена[30] в дизайне структуры данных. Для ситуации, когда требуется хранить метрики с сотен тысяч серверов с гранулярностью в одну минуту, это свойство также становится чрезвычайно важным;

3. Cassandra является высокодоступной базой данных. Благодаря скорости записи и механизму копирования данных сразу на несколько элементов кластера, Cassandra хорошо справляется с постоянным большим потоком входящих данных и снижает вероятность потери данных;

4. Для запросов в Cassandra используется SQL-подобный язык. SQL языки являются гибким и производительным инструментом работы с данными.

2.3.3 Anomalies detector

Детектор аномалий — это простой веб-сервис, принимающий данные в виде временных рядов и возвращающий список аномалий во введённых данных. Для реализации этого компонента подходит язык Python, поскольку он традиционно используется для работы с данными и имеет множество готовых инструментов. В качестве фреймворка для создания веб-приложений будет использоваться Flask[31].

2.3.4 Agent

Агенты — это легковесные daemon-процессы- приложения, которые будут устанавливаться на сервера, собирать метрики, накапливать их и перенаправлять на сервер мониторинга. Для этой задачи также подходят язык Kotlin и фреймворк Spring. Стек технологий для реализации данных приложений не принципиален.

2.3.5 Client

Клиентский модуль представляет собой библиотеку, которую можно подключить к приложению на любом языке программирования (на этом же языке должен быть написан и клиентский модуль). В рамках данной работы в качестве примера будет реализован клиентский модуль на Java.

2.4 Реализация

Общий алгоритм работы системы:

1. В приложение добавляется клиентская библиотека для мониторинга, предоставляющая готовые методы для отправки метрик;
2. Клиентское приложение при помощи библиотеки отправляет собранные метрики в агент по протоколу UDP;
3. Агент перенаправляет полученные данные в сервер мониторинга по протоколу UDP;
4. Сервер принимает полученные данные, подготавливает их к хранению и сохраняет в базу данных;
5. После этого администратор системы и внешние клиенты имеют возможность совершать запросы к серверу мониторинга для выполнения следующих действий:
   (a) выполнение выражений языка запросов (для прямого получения данных по метрикам или с предварительной обработкой);
   (b) создание алертов.

2.4.1 Monitoring server

В рамках выполнения выпускной квалификационной работы было реализовано приложение, выполняющее следующие ключевые функции:

1. Получение метрик через внешний интерфейс (по протоколу UDP);
2. Сохранение метрик в базу данных;
3. Обработка внешних запросов на получение данных;
4. Алертинг.

Получение метрик. Общий протокол отправки данных:

1. Клиент собирает текущее состояние по всем метрикам и с определённой периодичностью отправляет их на указанный в конфигурации клиента адрес (это может быть либо адрес агента, либо адрес самого сервера мониторинга);

2. Формат отправки данных следующий: содержимое сообщения представляет собой JSON[32] объект, содержащий два поля: поле «timestamp», определяющее время отправки метрик, и JSON массив с JSON объектами, каждый из которых содержит поля: идентификатор метрики (ключ в упомянутом в п. 2.2 формате) и значение метрики (числовые значения, которые преобразуются в Double);

3. Если данные были отправлены напрямую на сервер мониторинга, то данный пункт пропускается. Если данные были отправлены агенту, то агент их запоминает, буфферизует и отправляет на сервер мониторинга;

4. Сервер мониторинга слушает определяемый в конфигурации порт, ожидая сообщения, полученные по протоколу UDP. После этого приложение преобразует полученные данные и сохраняет их в базе данных.

Хранение метрик. Процесс сохранения полученных данных осуществляется в два этапа:

1. При получении сервером мониторинга новых данных эти данные сохраняются в базу данных без какой-либо обработки, в виде строки;

2. Данные, сохранённые в чистом виде, извлекаются из базы данных, преобразуются для структурированного хранения и сохраняются уже в обработанном виде.

Первый этап необходим для уменьшения потерь данных в ситуациях, когда в сервер мониторинга поступило слишком много запросов на сохра-

нение метрик. Перед сохранением в конечную хколлекцию, где хранятся метрики, необходимо обработать полученное сообщение – это действие предполагает некоторую нагрузку на систему. Поэтому используется буфер в виде коллекции «message_buffer» в базе данных, куда поступающее сообщение сохраняется моментально при получении сервером запроса. Коллекция «message_buffer» имеет два поля: «timestamp» (double) и «message» (string). Первое поле – время в формате Unix Timestamp[33], в которое было получено сообщение, второе поле – само сообщение. Время получения сообщения указывается с той целью, чтобы впоследствии можно было доставать сообщения в упорядоченном виде и обрабатывать в первую очередь самое старое сообщение, содержащееся в буфере. Такое решение принято исходя из предположения, что более старые сообщения содержат в себе более старые данные. Если же это по какой-то причине (например, из-за сетевых задержек) окажется не так, это не вызовет никаких ошибок – в сообщении содержится информация, необходимая для идентификации времени, когда был сделан замер для конкретной метрики.

На втором этапе отдельный процесс, работающий внутри сервера мониторинга, извлекает сообщения из коллекции-буфера, преобразует их для хранения и сохраняет в коллекцию «data_points_bucket». Эта коллекция является ключевой в вопросе хранения. Записи этой коллекции представляют сущность, которую можно назвать «бакетом» (от слова «ведро») – набор измерений метрики с определёнными метками, а также начальное и конечное время. Таким образом, при хранении измерения группируются в бакеты. Это нужно для ускорения запросов на получение данных: если хранить каждое измерение отдельно, то при поиске приходится проверять гораздо большее количество записей в коллекции. Все измерения для одной метрики попадают в один бакет в течение недели, если время нового измерения отличается от времени первого измерения в бакете на неделю или больше, то создаётся новый бакет.

Каждая запись состоит из пяти полей. В приведённом списке значение в кавычках – название поля, значение в скобках – тип данных[34].

1. «metric_name» (string) – имя метрики;

2. «labels» (map<string, string>) – коллекция меток (ключ – название метки, значение – значение метки);

3. «measurements» (map<timestamp, double>) – коллекция измерений (ключ – время в формате Unix Timestamp, значение – значение метрики в заданный момент времени);

4. «start_time» (timestamp) – начальное время бакета;

5. «end_time» (timestamp) – конечное время бакета.

Обработка запросов на получение данных. Было разработано REST API для получения данных о собранных метриках. Ключевой частью API является интерфейс для отправки в систему выражений на разработанном языке запросов. Формат предполагает возможность использования в запросе идентификаторов метрик и предопределённых функций.

Пример такого выражения:
«abs(metric_namelabel1=value1,label2=value2)». Запрос с таким выражением вернёт список тех значений метрики «metric_name», для которых значение метки «label1» это «value1», метки «label2» - «value2», при этом от каждого значения метрики будет взят модуль.

Результатом запроса может быть одна из двух структур:

1. Вектор;

2. Числовое значение.

Далее представлен список предопределённых функций. В скобках после названия каждой функции указан список аргументов, которые функция принимает. Первый аргумент каждой функции – «expression» – это любое выражение на языке запросов.

1. abs(expression) – взятие модуля. Если выражение в скобках является вектором, то каждое числовое значение вектора будет заменено на модуль от этого значения; если числовым значением, то будет возвращён модуль этого значения;

2. min(expression) — вычисление минимального значения вектора. Принимает только вектор, возвращает минимальное значение;

3. max(expression) – вычисление максимального значения вектора. Принимает только вектор, возвращает максимальное значение;

4. ceiling(expression) – округление вниз. Если выражение в скобках является вектором, то каждое числовое значение вектора будет округлено вниз к ближайшему целому; если числовым значением, то оно будет округлено вниз к ближайшему целому;

5. flour(expression) – округление вверх. Если выражение в скобках является вектором, то каждое числовое значение вектора будет округлено вверх к ближайшему целому; если числовым значением, то оно будет округлено вверх к ближайшему целому;

6. delta(expression) – вычисление разницы между начальным и конечным значением вектора. Принимает только вектор. Разница между первым и последним значением вектора;

7. sqrt(expression) — квадратный корень. Если выражение в скобках является вектором, то будет взят квадратный корень от каждого значения вектора; если числовым значением, то будет взят квадратный корень от этого значения;

8. sort(expression) — сортировка по возрастающей. Принимает только вектор. Сортирует элементы временного ряда по возрастанию значения;

9. sortDescending(expression) — сортировка по убывающей. Принимает только вектор. Сортирует элементы временного ряда по убыванию значения;

10. predict(expression, prediction_time) – предсказание будущих значений ряда. Принимает вектор, а также время, на которое нужно построить предсказание. Возвращает вектор, расширенный предсказанными значениями.

Предсказание будущих значений. В рамках задачи мониторинга предсказание будущих значений временного ряда может быть полезно для того, чтобы узнать о потенциальной проблеме в системе до её появления. Таким образом, можно настроить на определённую метрику алерт, который будет вычислять будущее значение метрики и сообщать администратору о том, что возникла потенциально опасная ситуация и может потребоваться ручной контроль.

Важными параметрами анализа временного ряда являются тренд и сезонность. Тренд – это некоторая общая тенденция, которой подвержены уже известные значения ряда. Сезонность – это влияние периодичности на значения ряда.

Существует множество алгоритмов предсказания значений временного ряда. В разрабатываемой системе для предсказания был использован метод экспоненциального сглаживания. Существуют также различные подвиды этого метода. Ключевыми подвидами являются:

1. Одиночное экспоненциальное сглаживание. Данный подвид не учитывает тренд и сезонность;

2. Двойное экспоненциальное сглаживание. Данный подвид учитывает тренд, но не учитывает сезонность;

3. Тройное экспоненциальное сглаживание. Данный подвид учитывает и тренд, и сезонность;

В связи с тем, что метод тройного экспоненциального сглаживания (также иногда называемый методом Холта-Уинтерса[35]) учитывает сезонность и тренд, что может быть полезно учитывать при анализе метрик (например, потому, что нагрузка на рассматриваемые сервисы может повышаться или понижаться в определённое время дня или года), было принято решение использовать именно этот метод.

Алгоритм выглядит следующим образом:

Общее сглаживание:

$$S_t = \alpha \frac{y_t}{I_{t-L}} + (1 - \alpha)(S_{t-1} + b_{t-1})$$

Сглаживание тенденции:

$$b_t = \gamma(S_t - S_{t-1}) + (1 - \gamma)b_{t-1}$$

Сглаживание сезонности:

$$I_t = \beta\frac{y_t}{S_t} + (1 - \beta)T_{t-L}$$

Прогноз: (t+m-ое наблюдение):

$$F_{t+m} = (S_t + mb_t)I_{t-L+m}$$

Где:

y - наблюдение;

α - коэффициент сглаживания для уровня;

β - коэффициент сглаживания тренда;

γ - коэффициент сглаживания сезонности;

m - количество точек, для которых будет вычислен прогноз.

Последовательность действий:

1. По формуле (1) вычисляются начальные (для t = 1) значения сглаживания сезонности и тренда;

$$b = \frac{1}{L}(\frac{y_{L+1} - y_1}{L} + \frac{y_{L+2} - y_2}{L} + ... + \frac{y_{L+L} - y_L}{L}) \qquad (1)$$

2. Зная эти начальные значения, можно вычислить общее сглаживание, сглаживание сезонности и тренда для следующего t;

3. Последовательно вычисляются следующие значения общего сглаживания, сглаживания сезонности, тренда и прогноза значения вплоть до t + m.

Алертинг. В системе существует модуль, отвечающий за алертинг. Основной сущностью является конфигурация алерта — сущность, свойства которой определяют, в какой ситуации сработает триггер (условие срабатывания) алерта и будут отправлены оповещения. Конфигурации алертов описаны в файле конфигурации, который лежит на диске сервера.

Алерты объединены в группы (это нужно как для логического выделения групп близких алертов, так и для практических целей: например, можно настроить оповещение определённого круга пользователей только для конкретных групп алертов). Каждая группа имеет имя и список алертов, ассоциированный с ней. Каждый алерт может принадлежать только к одной группе. У каждого алерта указывается условие выполнение, длительность (время, в течение которого должно выполняться условие) и сообщение, которое будет отправлено администраторам системы в случае, если условие будет выполняться в течение заданного времени.

Пример файла конфигурации:

```
alert-groups:
    -
    name: alert-group1
    alerts:
      -
        for: 5m
        condition: predefined_anomaly_detection(metric_name{
        label1=value1,label2=value2},
        ["bitmap_detector", "default_detector",
        "derivative_detector", "exp_avg_detector"])
        message: Found an anomaly!
  -
    name: alert-group2
    alerts:
      -
        for: 15m
        condition: avg(metric_name{label1_value1,
```

```
label2=value2}) > 3.0
message: Average of metric is larger than the border
over the last 5 minutes.
```

Важно обратить внимание: алерт из группы «alert-group1» использует функцию «predefined_anomaly_detection» - эта функция не была описана в списке функций языка запросов. Дело в том, что эта функция доступна только для алертов. Она принимает два аргумента: временной ряд (определяемый либо идентификатором метрики, либо любым выражением на языке запросов, возвращающим временной ряд) и список алгоритмов обнаружения аномалий. По умолчанию в списке два метода: «bitmap_detector», «default_detector».

2.4.2 Обнаружение аномалий

Сервис обнаружения аномалий представляет собой приложение, написанное на Python с использованием фреймворка для веб-приложений Flask. Для обнаружения аномалий используется luminol[36] — библиотека, выполняющая две ключевые функции: обнаружение аномалий во временных рядах и определение корреляции временных рядов. Компонент anomalies detector принимает временные ряды и список алгоритмов, которые будут использованы для поиска аномалий. В случае, если переданный список пуст, компонент использует все алгоритмы из списка и вернёт все возможные аномалии. Из предоставляемых библиотекой luminol алгоритмов на данный момент используются:

1. bitmap_detector — алгоритм, основывающийся на битовых картах; данный алгоритм разбивает временной ряд на множество частей и использует частоту схожих частей для вычисления вероятности присутствия аномалии в участке;

2. derivative_detector — алгоритм, который высчитывает ряд производных точек исходного ряда и рассматривает точки этого ряда в сравнении со скользящим средним от него же;

3. exp_avg_detector — этот алгоритм для каждой точки высчитывает скользящее среднее «окна отсрочки» - множества точек ряда, предшествующих текущей (по умолчанию размер окна вычисляется как число 0.2, умноженное на размер переданного временного ряда); модуль разницы значения точки и скользящего среднего для неё он считает вероятностью присутствия аномалии в данной точке;

4. default_detector — в данном алгоритме используются данные, полученные от двух предыдущих алгоритмов, проводится серия сравнений между ними и выбираются наиболее подходящие результаты.

Ответ сервиса обнаружения аномалий содержит в себе набор предполагаемых аномалий для каждого алгоритма. У каждой аномалии существует четыре поля - «очки» (чем выше количество очков, тем значительнее аномалия), время начала и конца наблюдения аномалии во временном ряду, и предположительное время аномалии.

Пример ответа сервиса:


```
{
  "anom": {
    "bitmap_detector": [
      {
        "anomaly_score": 10.238565727249224,
        "end_timestamp": 1388874840000,
        "exact_timestamp": 1388874660000,
        "start_timestamp": 1388874600000
      },
      {
        "anomaly_score": 5.335397664272317,
        "end_timestamp": 1388880060000,
        "exact_timestamp": 1388880060000,
        "start_timestamp": 1388880000000
      }
    ],
```

```
    "default_detector": [
      {
        "anomaly_score": 10.238565727249224,
        "end_timestamp": 1388874840000,
        "exact_timestamp": 1388874660000,
        "start_timestamp": 1388874600000
      },
      {
        "anomaly_score": 5.335397664272317,
        "end_timestamp": 1388880060000,
        "exact_timestamp": 1388880060000,
        "start_timestamp": 1388880000000
      }
    ]
}
```


2.4.3 Клиенты

У сервиса мониторинга может быть множество различных источников. В контексте данной работы это могут быть веб-приложения или некоторые показатели серверов, на которых они работают (например, количество свободной памяти на диске или загрузка процессора). В рамках работы была разработана клиентская библиотека на языке Java, ключевым интерфейсом которой является регистр метрик — объект, в котором приложение «регистрирует» метрики (объявляет имена метрик и их метки). Затем приложение, по мере получения данных, пишет эти метрики в регистр, а отдельный поток, называемый «отправщиком», с определённым интервалом забирает данные из регистра и отправляет их по адресу, указанному в конфигурации приложения (по этому адресу может находиться либо сервер мониторинга, либо агент).

Клиентские библиотеки можно реализовать и на других языках программирования, в зависимости от потребностей определённой системы.

Помимо прочего, можно написать специальные приложения, которые

будут экспортировать статистически важные данные из уже созданных приложений (например, из баз данных) или с самих серверов. В данной работе это рассматриваться не будет.

2.4.4 Агенты

Агенты были реализованы на языке Kotlin с использованием фреймворка Spring. Это небольшие легковесные приложения, выступающие буфером между клиентскими приложениями и сервером мониторинга. Клиентские приложения отправляют метрики по протоколу UDP в агенты, агенты в свою очередь запоминают их и отправляют в сервер мониторинга.

Глава 3. Тестирование

В рамках выполнения выпускной квалификационной работы было выделено три этапа тестирования разработанного продукта:

1. Ручное интеграционное функциональное тестирование;
2. Автоматизированное интеграционное функциональное тестирование;
3. Нагрузочное тестирование.

3.1 Ручное интеграционное функциональное тестирование

Ручное тестирование предполагает моделирование поведение пользователя (или клиентских приложений) при помощи различных утилит. В случае с тестированием веб-приложения, у которого есть графический интерфейс, для тестирования также может использоваться веб-браузер[37]. Ручное тестирование полезно для тех ситуаций, когда при тестировании необходимо выполнить действия, которые сложно автоматизировать. Также чаще всего те действия, которые планируется автоматизировать, сначала выполняются вручную: для того, чтобы чётко выделить шаги тестирования, а также для того, чтобы удостовериться, что автоматизированный тест не выдаёт ложно-положительный результат.

Для ручного тестирования разработанного приложения использовались следующие инструменты:

1. curl[38] – утилита командной строки, позволяющая передавать данные по различным протоколам передачи данных. Данная утилита использовалась для совершения запросов к системе (в частности, запросов, получающих от сервера результаты выполнения различных выражений на языке запросов) по протоколу HTTP;
2. netcat[39] – утилита командной строки, позволяющая устанавливать UDP и TCP соединения. Была использована для отправки сообщения

на сервер мониторинга по протоколу UDP с целью имитации поведения клиентских приложений.

Основной функционал, протестированный в рамках ручного тестирования:

1. Получение метрик сервером мониторинга. Для тестирования данной функции использовалась утилита netstat. При помощи данной утилиты генерировались UDP-пакеты и отправлялись на порт, который слушал сервер мониторинга. В силу того, что протокол UDP не предполагает получения клиентом ответа от сервера, способность сервера мониторинга получать данные проверялась в рамках п. 3. Пример запроса в командной строке Linux:

```
$ echo -n "{"data": [{"id": "requests_number
{server=test-server.com,path=/api/users}",
"values": [{"1587510432000": 12.0,
"1587510492000": 29.0, "1587510552000": 97.0}]}]}"
| nc -4u -w0 localhost 8000
```

2. Обработка сервером мониторинга запросов к вспомогательным методам API сервера мониторинга (в частности, к методам, возвращающим списки имён и меток сохранённых в базе данных сервера метрик). Данные методы полезно использовать для того, чтобы получить представление об идентификаторах имеющихся метрик с целью использования этих идентификаторов в запросах на языке запросов. Пример запроса в командной строке Linux:

```
$ curl localhost:8000/api/metrics
```

Пример ответа:

```
["requests_number"]
```

Пример запроса для меток:

```
$ curl localhost:8000/api/metrics/requests_number/labels
```

Пример ответа:

```
["server", "path"]
```

3. Обработка сервером мониторинга запросов на языке запросов. При помощи утилиты curl были совершены запросы к серверу мониторинга, содержащие в себе выражения на языке запросов, и проверялось соответствие запроса полученному ответу. Также полученные в результате обработки запросов данные сравнивались со введёнными в п. 1 данными – они должны были быть равными. Пример запроса в командной строке Linux:

```
$ curl localhost:8000/api/query
?query=requests_number%7Bserver=
test-server.com,path=/api/users%7D
```

Пример ответа:

```
{"type": "VECTOR", "data": {"requests_number
{server=test-server.com,path=/api/users}":
[{"1587510432000": 12.0, "1587510492000": 29.0,
"1587510552000": 97.0}]}}
```

Пример запроса с функцией языка запросов:

```
curl localhost:8000/api/query
?query=avg(requests_number%7Bserver
=test-server.com,path=/api/users%7D)
```

Пример ответа:

```
{"type": "DOUBLE", "data": 46.0}
```

3.2 Автоматизированное интеграционное функциональное тестирование

Автоматизированное тестирование – это аналог ручного тестирования, в котором действия, необходимые для тестирования, выполняются не человеком, а программой. Преимущества автоматизированного тестирования:

1. Качество тестирования. Автоматизирование тестирования избавляет результаты теста от влияния человеческого фактора;
2. Ускорение процесса тестирования. Автоматизированный тест требует времени на написание, но время выполнения автоматизированного теста всегда значительно ниже времени выполнения того же теста вручную;
3. Возможность выполнения тех видов тестов, которые невозможно либо затруднительно выполнить вручную;

Были реализованы автоматизированные blackbox тесты, проверяющие основной функционал системы. Данные тесты повторяют сценарии проверок, выполненных в рамках ручного тестирования. Таким образом, автоматизированные тесты проверяют возможность отправить данные на сервер мониторинга, после их сохранения получить метаинформацию (имена метрик и меток) и сами данные в чистом и агрегированном виде (при помощи выражений на языке запросов).

Основным преимуществом автоматизированных тестов является возможность быстро проводить регрессионное тестирование после внесения изменений в систему для того, чтобы убедиться, что внесённые изменения не оказали негативного влияния на уже разработанный и протестированный функционал системы. Регрессионное тестирование проводилось непосредственно после внесения изменений в компоненты системы на финальном этапе разработки системы. Таким образом, написанные автоматизированные тесты избавляли от необходимости проводить ручное после каждого изменения. В дальнейшем реализованные автоматизированные тесты также могут выполнять данную функцию, экономя время на доработку системы и уменьшая вероятность допустить ошибку при разработке. За счёт того, что сценарии тестов предполагают именно тестирование методом чёрного ящика, то есть программа, осуществляющая тесты, взаимодействует с тестируемым приложением при помощи внешних интерфейсов и имитирует поведение реального пользователя, разработанные тесты также могут быть полезны при проверке корректности проведённой установки системы мониторинга на новое окружение. После того, как система была развёрнута на новой группе серверов, администратор, проводивший установку, даже не имея знаний об устройстве системы может запустить автоматизированные тесты, которые могут показать ему, является ли установленная система готовой к эксплуатации.

3.3 Нагрузочное тестирование

В рамках выполнения выпускной квалификационной работы также было проведено нагрузочное тестирование. Нагрузочное тестирование это подвид тестирования, в рамках которого путём выполнения большого количества запросов в сжатое время проверяются производительность и отказоустойчивость системы. Данный вид тестирования необходим для:

1. Определения максимального количества пользователей, которые могут работать с приложением одновременно;

2. Определения времени отклика приложения;

3. Поиска и устранение неоптимальных решений в разработке.

Для осуществления нагрузочного тестирования использовался инструмент Apache Jmeter[40] – программное обеспечение, разработанное для нагрузочного тестирования веб-приложений. Этот инструмент позволяет создавать сценарии тестирования и запускать их с разными параметрами (например, длительность сценария или количество одновременных пользователей).

Нагрузочное тестирование выполнялось посредством вызова метода API, осуществляющего обработку запросов на языке запросов.

Система мониторинга была развёрнута на сервере со следующей конфигурацией: 8 ядер CPU[41], 16 гигабайт оперативной памяти, 250 гигабайт SSD[42].

Далее приведены таблицы с результатами тестирования. Характеристики результата (всё время указывается в миллисекундах):

1. Samples – количество вызовов, совершённых во время тестирования;
2. Error – процент ошибок;
3. Average – среднее время обработки запроса;
4. Min – минимальное время, за которое был обработан запрос;
5. Max – максимальное время, за которое был обработан запрос;
6. 90th pct, 95th pct, 99th pct – 90%-ный, 95%-ный и 99%-ый перцентили соответственно. За это либо меньшее время было обработано 90%, 95% и 99% запросов соответственно;
7. Transactions/s – количество запросов в секунду.

20 пользователей:

Samples	18730
Error	0.00%

Average	407.47
Min	175.91
Max	1952.78
90th pct	394.04
95th pct	479.56
99th pct	1525.46
Transaction/s	31.18

Таблица 1: Результаты тестирования

40 пользователей:

Samples	11049
Error	0.79%
Average	1866.40
Min	159.17
Max	4611.23
90th pct	2053.11
95th pct	2231.89
99th pct	2599.93
Transaction/s	16.36

Таблица 2: Результаты тестирования

В случае с 40 пользователями можно заметить, что произошло резкое снижение производительности , а также количество ошибок стало ненулевым, хотя нагрузка возросла всего в два раза. Это указывает на то, что при таком количестве параллельных потоков, совершающих запросы без остановки, система мониторинга, развёрнутая в единственном экземпляре на сервере с указанной выше конфигурацией, перестаёт справляться с на-

грузкой. Таким образом, обнаружена граница максимального количества одновременно работающих пользователей, которых может обслуживать система.

Глава 4. Заключение

В рамках выполнения выпускной квалификационной работы была сформулирована задача построения системы мониторинга. Проведён анализ существующих решений в сфере мониторинга, на основе которого были сформулированы основные функциональные требования, к поставленной задаче и учитывающие требования к современным веб-приложениям. Спроектирована архитектура решения и выбран технологический стек, позволяющий эффективно и быстро решить поставленную задачу.

Разработаны все ключевые модули системы. Проведён и описан процесс функционального и нагрузочного тестирования. Нагрузочное тестирование показало результаты, которые можно считать удовлетворительными для современной системы мониторинга.

Разработанная система учитывает преимущества и недостатки существующих решений в сфере мониторинга, и её основными преимуществами являются простота использования, низкая стоимость мониторинга для клиентских приложений, гибкий язык запросов, позволяющий извлекать и агрегировать данные, использование алгоритмов поиска аномалий в данных метрик.

Исходный код компонентов системы доступен по следующим ссылкам:

1. Сервер мониторинга[43];
2. Клиентская библиотека для языка Java[44];
3. Агент[45];
4. Сервис обнаружения аномалий[46].

Список литературы

[1] Определение мониторинга.
https://en.wikipedia.org/wiki/Website_monitoring

[2] Славек Лигус, Effective Monitoring and Alerting: For Web Operations, 2012

[3] Майк Джулиан, Practical Monitoring: Effective Strategies for the Real World, 2017

[4] В.П. Шкодырев, К.И. Ягафаров, В.А. Баштовенко, Е.Э. Ильина, Обзор методов обнаружения аномалий в потоках данных, 2017

[5] А.А. Пивень, Ю.И. Скорин, Тестирование программного обеспечения, 2012

[6] Р.А. Нагаев, И.С. Полевщиков, Автоматизация процесса тестирования программного обеспечения с применением Junit, 2016

[7] Документация collectd. https://collectd.org/documentation.shtml

[8] Описание формата Whisper.
https://graphite.readthedocs.io/en/latest/whisper.html

[9] Документация InfluxDB. https://docs.influxdata.com/influxdb

[10] LSM-дерево.
https://ru.wikipedia.org/wiki/LSM-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE

[11] TSM-дерево.
https://docs.influxdata.com/influxdb/v1.7/concepts/storage_engine/

[12] Документация OpenTSDB. http://opentsdb.net/docs/build/html/

[13] Описание TCollector.
http://opentsdb.net/docs/build/html/user_guide/utilities/tcollector.html

[14] Описание time series daemon. http://opentsdb.net/overview.html

[15] Документация Hadoop. https://hadoop.apache.org/docs/stable/

[16] Документация HBase. https://hbase.apache.org/book.html

[17] Документация Nagios. https://www.nagios.org/documentation/

[18] Документация RabbitMQ. https://www.rabbitmq.com/documentation.html

[19] Документация Redis. https://redis.io/documentation

[20] Спецификация протокола UDP. https://tools.ietf.org/html/rfc768

[21] Документация Kotlin. https://kotlinlang.org/

[22] Описание работы JVM. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html

[23] Null-безопасность в Kotlin. https://kotlinlang.org/docs/reference/null-safety.html

[24] Корутины в Kotlin. https://kotlinlang.org/docs/reference/coroutines-overview.html

[25] Документация Spring Framework. https://spring.io/projects/spring-framework

[26] Описание механизма Dependency injection. https://en.wikipedia.org/wiki/Dependency_injection

[27] Документация Cassandra. https://cassandra.apache.org/

[28] NoSQL базы данных. https://en.wikipedia.org/wiki/NoSQL

[29] Принципы ACID. https://en.wikipedia.org/wiki/ACID

[30] Формат данных в Cassandra. https://cassandra.apache.org/doc/latest/data_modeling/intro.html

[31] Документация Flask. https://flask.palletsprojects.com/en/1.1.x/

[32] Формат данных JSON. https://www.json.org/json-en.html

[33] Unix Timestamp. https://tools.ietf.org/html/rfc3339

[34] Типы данных в Cassandra.
https://cassandra.apache.org/doc/latest/cql/types.html

[35] Модель Холта-Уинтерса.
https://upcommons.upc.edu/bitstream/handle/2117/120562/LOAD%20FORECASTING%20USING%20HOLT-WINTERS%20METHOD.pdf

[36] https://github.com/linkedin/luminol. Библиотека luminol.

[37] Определение веб-браузера.
https://en.wikipedia.org/wiki/Web_browser

[38] Документация cURL. https://curl.haxx.se/docs/

[39] Документация netcat. http://netcat.sourceforge.net/

[40] Документация Jmeter. https://jmeter.apache.org/

[41] CPU. https://en.wikipedia.org/wiki/Central_processing_unit

[42] SSD. https://en.wikipedia.org/wiki/Solid-state_drive

[43] Репозиторий, содержащий исходный код сервера мониторинга. https://github.com/purrfessor/monitoring

[44] Репозиторий, содержащий исходный код клиентской библиотеки для языка Java. https://github.com/purrfessor/monitoring-client

[45] Репозиторий, содержащий исходный код агента мониторинга. https://github.com/purrfessor/monitoring-agent

[46] Репозиторий, содержащий исходный код сервиса обнаружения аномалий. https://github.com/purrfessor/anomalies-detector